What is Activity Diagram?

visual-paradigm.com/guide/uml-unified-modeling-language/what-is-activity-diagram/

Activity diagram is another important behavioral diagram in UML diagram to describe dynamic aspects of the system. Activity diagram is essentially an advanced version of flow chart that modeling the flow from one activity to another activity.

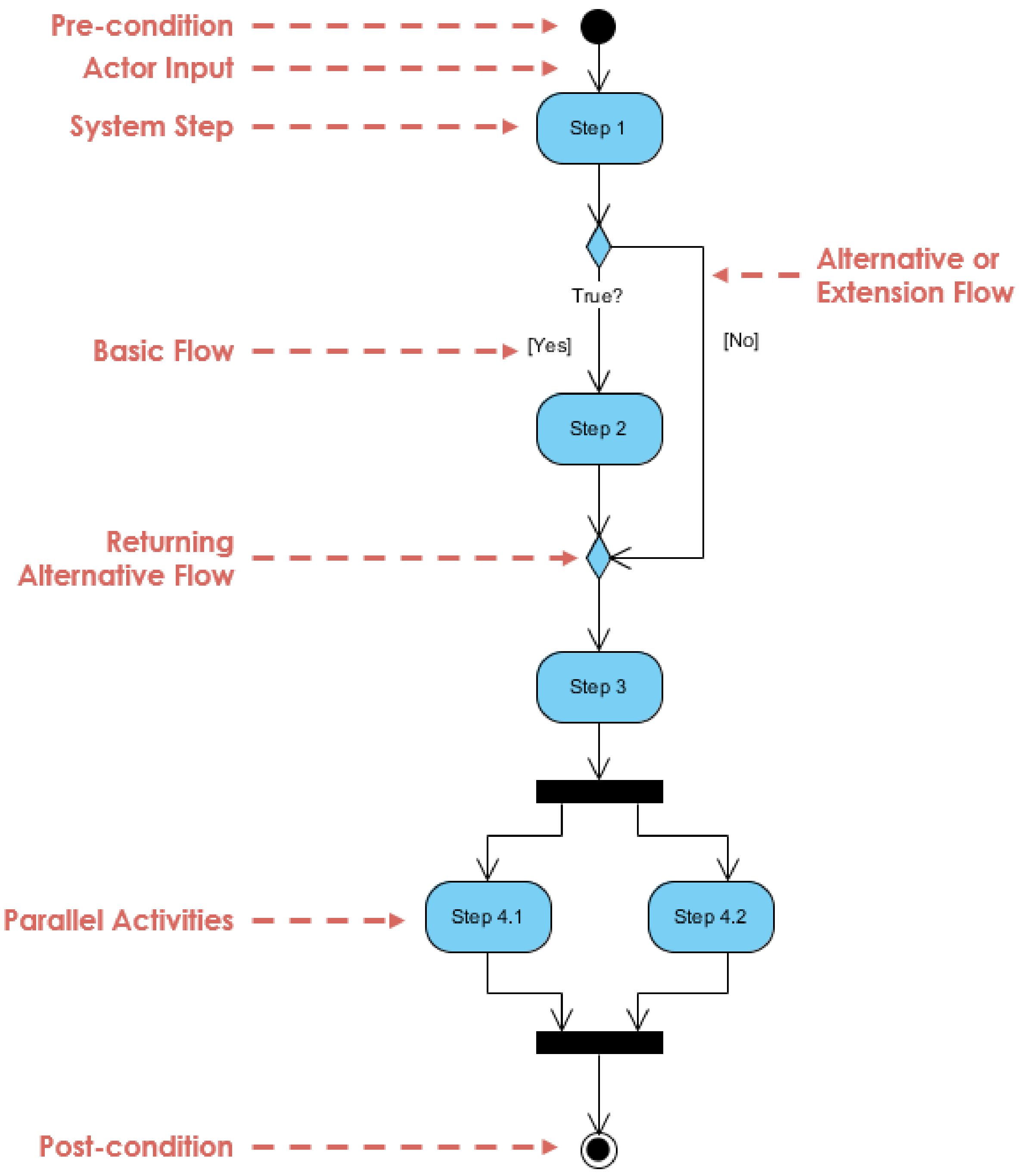
When to Use Activity Diagram

Activity Diagrams describe how activities are coordinated to provide a service which can be at different levels of abstraction. Typically, an event needs to be achieved by some operations, particularly where the operation is intended to achieve a number of different things that require coordination, or how the events in a single use case relate to one another, in particular, use cases where activities may overlap and require coordination. It is also suitable for modeling how a collection of use cases coordinate to represent business workflows

1. Identify candidate use cases, through the examination of business workflows
2. Identify pre- and post-conditions (the context) for use cases
3. Model workflows between/within use cases
4. Model complex workflows in operations on objects
5. Model in detail complex activities in a high level activity Diagram

Activity Diagram - Learn by Examples

A basic activity diagram - flowchart like

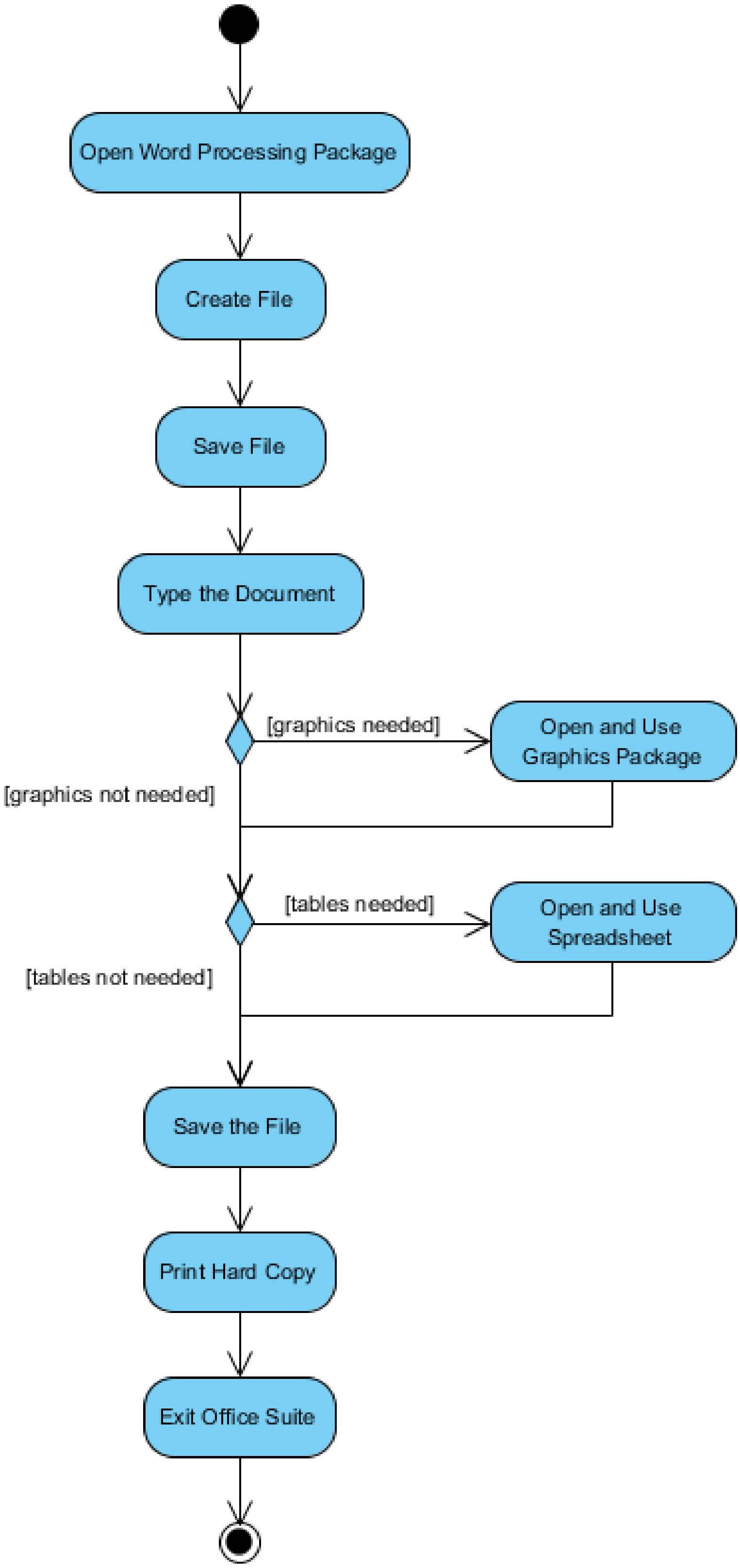


Activity Diagram - Modeling a Word Processor

The activity diagram example below describes the workflow for a word process to create a document through the following steps:

- Open the word processing package.
- Create a file.
- Save the file under a unique name within its directory.
- Type the document.
- If graphics are necessary, open the graphics package, create the graphics, and paste the graphics into the document.
- If a spreadsheet is necessary, open the spreadsheet package, create the spreadsheet, and paste the spreadsheet into the document.
- Save the file.

- Print a hard copy of the document.
- Exit the word processing package.



Activity Diagram Example - Process Order

Given the problem description related to the workflow for processing an order, let's model the description in visual representation using an activity diagram:

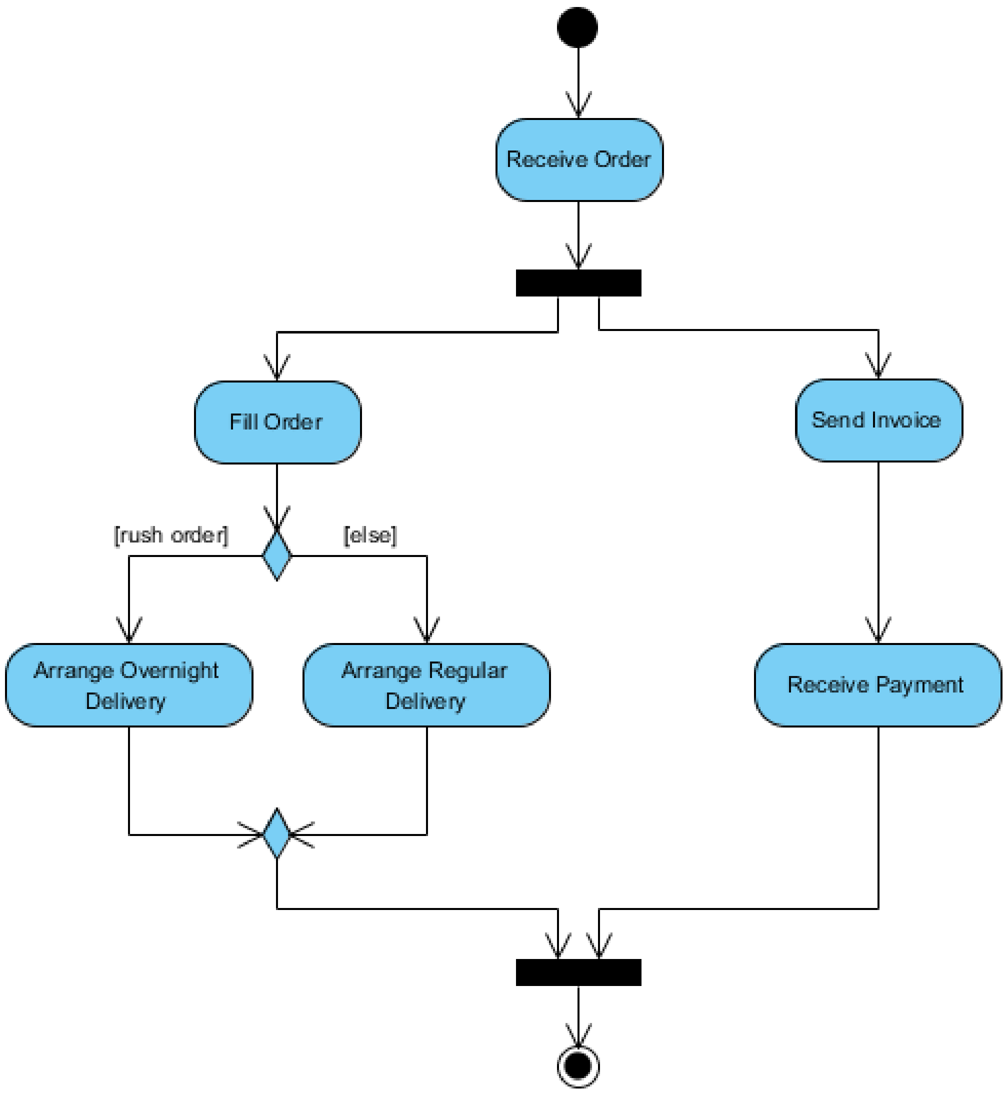
Process Order - Problem Description

Once the order is received, the activities split into two parallel sets of activities. One side fills and sends the order while the other handles the billing.

On the Fill Order side, the method of delivery is decided conditionally. Depending on the condition either the Overnight Delivery activity or the Regular Delivery activity is performed.

Finally the parallel activities combine to close the order.

The activity diagram example below visualize the flow in graphical form.

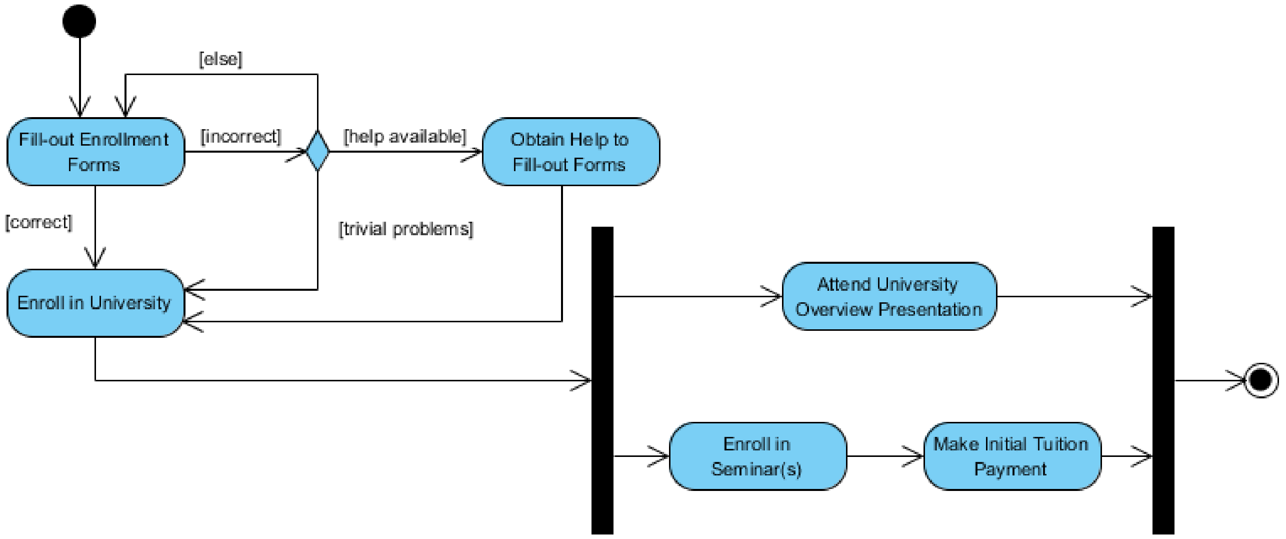


Activity Diagram Example - Student Enrollment

This UML activity diagram example describes a process for student enrollment in a university as follows:

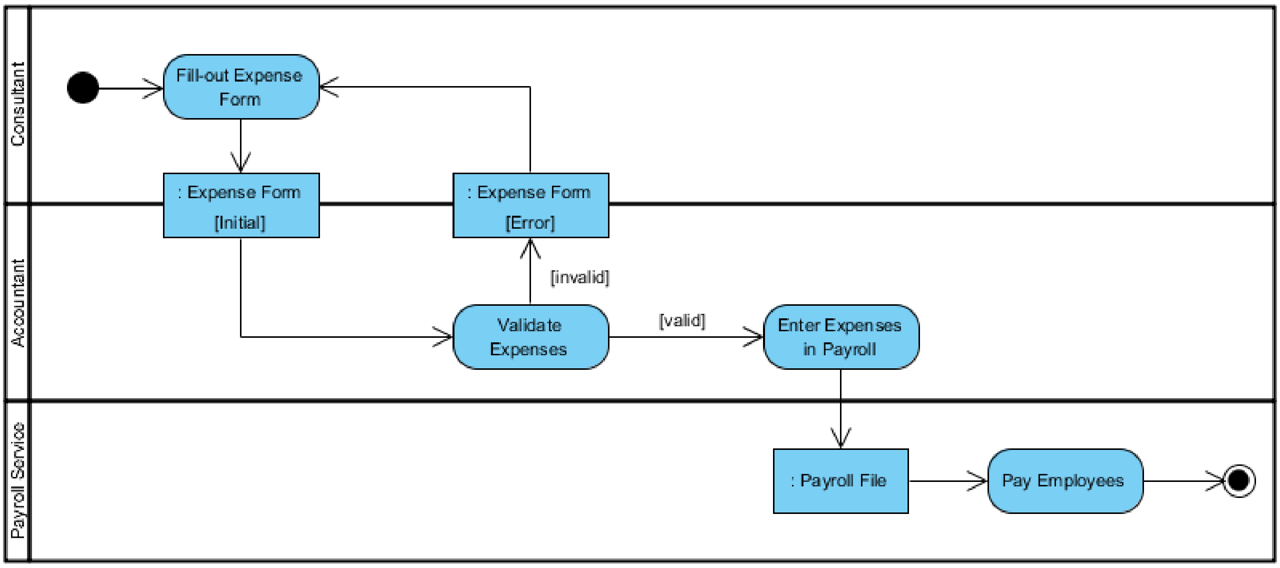
- An applicant wants to enroll in the university.
- The applicant hands a filled out copy of Enrollment Form.
- The registrar inspects the forms.

- The registrar determines that the forms have been filled out properly.
- The registrar informs student to attend in university overview presentation.
- The registrar helps the student to enroll in seminars
- The registrar asks the student to pay for the initial tuition.


[else]
Fill-out Enrollment Forms
[incorrect]
[help available]
Obtain Help to Fill-out Forms
[correct]
[trivial problems]
Enroll in University
Attend University Overview Presentation
Enroll in Seminar(s)
Make Initial Tuition Payment

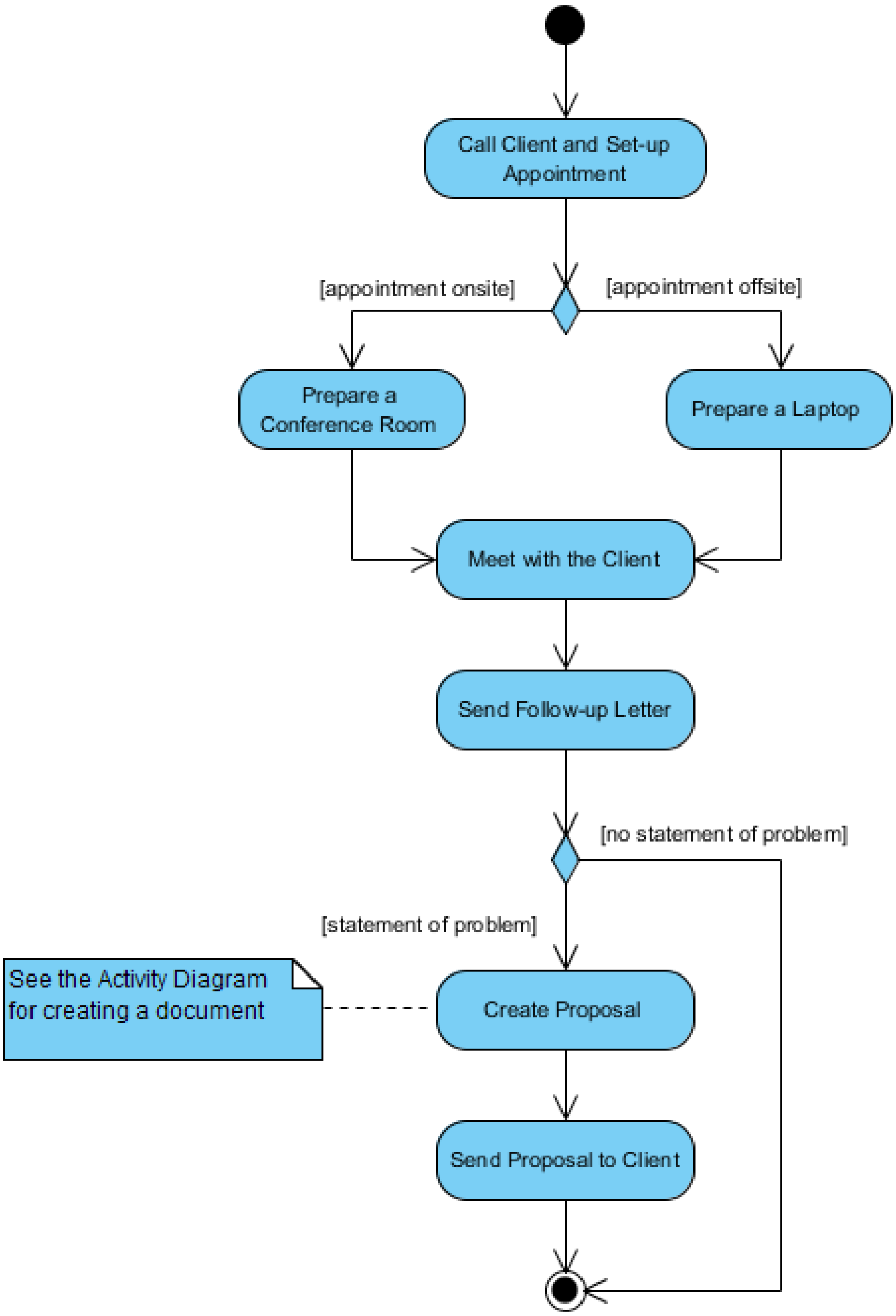
Activity Diagram - Swinlane

A swimlane is a way to group activities performed by the same actor on an activity diagram or activity diagram or to group activities in a single thread. Here is an example of a swinlane activity diagram for modeling Staff Expenses Submission:

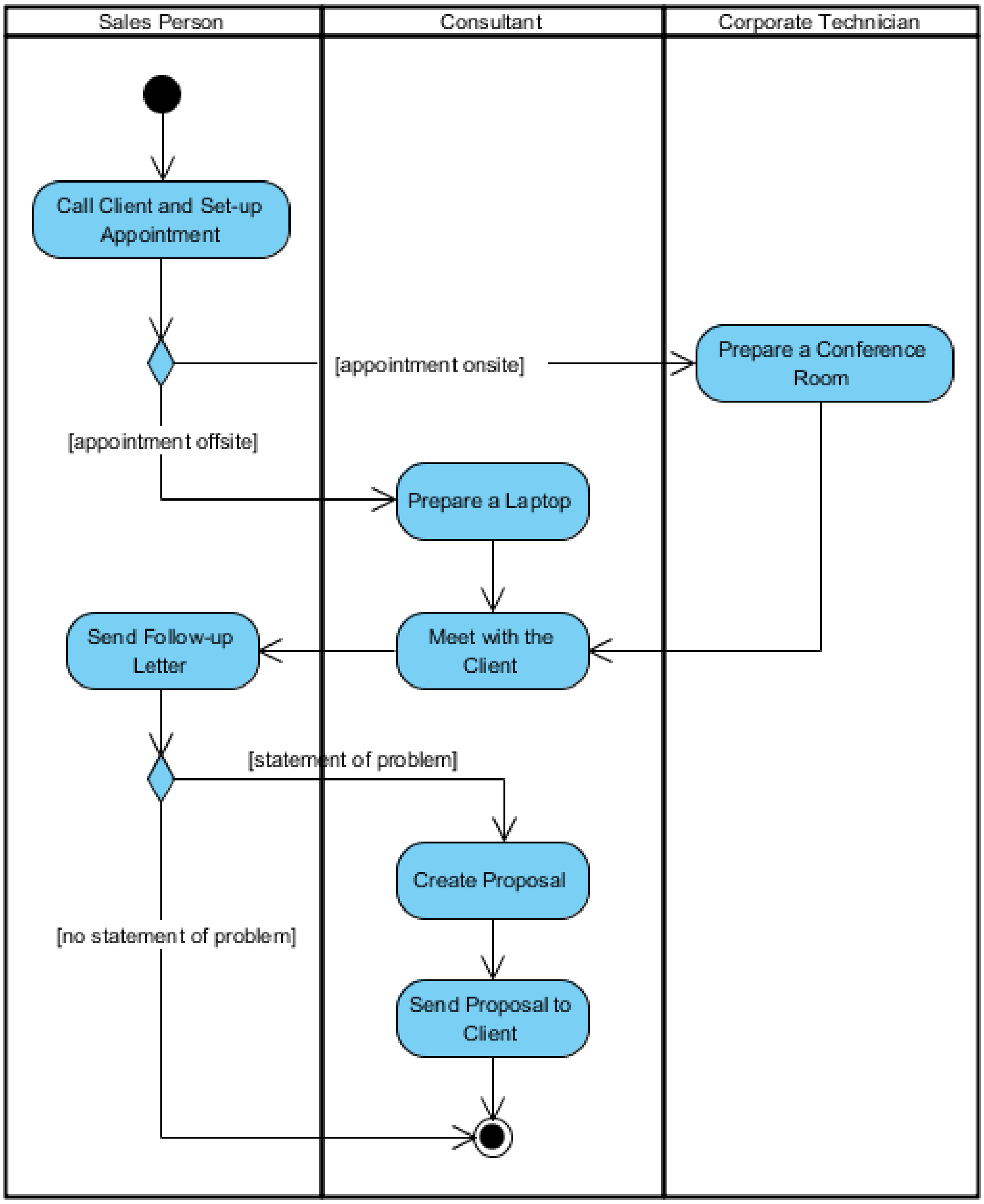


Swinlane and Non-Swinlane Activity Diagram

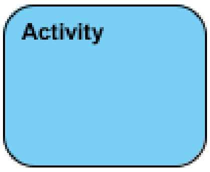
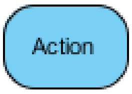
The activity diagram example below describes the business process for meeting a new client using an activity Diagram without swinlane.



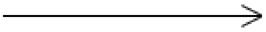
This figure below describes the business process for meeting a new client using an activity Diagram with swinlane.



Activity Diagram Notation Summary

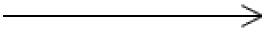
Notation Description	UML Notation
Activity Is used to represent a set of actions	
Action A task to be performed	

Control Flow



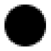
Shows the sequence of execution

Object Flow



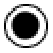
Show the flow of an object from one activity (or action) to another activity (or action).

Initial Node



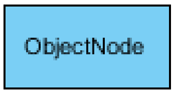
Portrays the beginning of a set of actions or activities

Activity Final Node



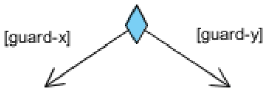
Stop all control flows and object flows in an activity (or action)

Object Node



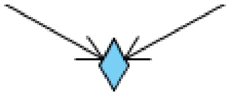
Represent an object that is connected to a set of Object Flows

Decision Node



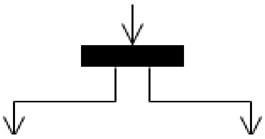
Represent a test condition to ensure that the control flow or object flow only goes down one path

Merge Node



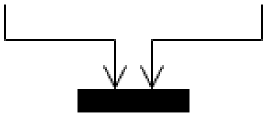
Bring back together different decision paths that were created using a decision-node.

Fork Node



Split behavior into a set of parallel or concurrent flows of activities (or actions)

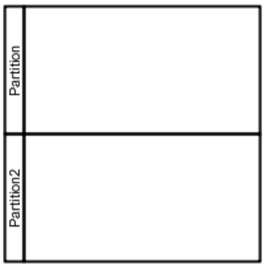
Join Node



Bring back together a set of parallel or concurrent flows of activities (or actions).

Swimlane and Partition

A way to group activities performed by the same actor on an activity diagram or to group activities in a single thread



What is Use Case Diagram?

visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/

Here are some questions that have been asked frequently in the UML world are: **What is a use case diagram? Why Use case diagram?** or simply, **Why use cases?**. Some people don't know what use case is, while the rest under-estimated the usefulness of use cases in developing a good software product. Is use case diagram underrated? I hope you will find the answer when finished reading this article.

So what is a use case diagram? A UML use case diagram is the primary form of system/software requirements for a new software program underdeveloped. Use cases specify the expected behavior (what), and not the exact method of making it happen (how). Use cases once specified can be denoted both textual and visual representation (i.e. use case diagram). A key concept of use case modeling is that it helps us design a system from the end user's perspective. It is an effective technique for communicating system behavior in the user's terms by specifying all externally visible system behavior.

A use case diagram is usually simple. It does not show the detail of the use cases:

- It only summarizes **some of the relationships** between use cases, actors, and systems.
- It does **not show the order** in which steps are performed to achieve the goals of each use case.

As said, a use case diagram should be simple and contains only a few shapes. If yours contain more than 20 use cases, you are probably misusing use case diagram.

The figure below shows the UML diagram hierarchy and the positioning of the UML Use Case Diagram. As you can see, use case diagrams belong to the family of behavioral diagrams.

Note that:

- There are many different UML diagrams that serve different purposes (as you can see from the UML diagram tree above). You can describe those details in other UML diagram types and documents, and have them be linked from use cases.
- Use cases represent only the functional requirements of a system. Other requirements such as business rules, quality of service requirements, and implementation constraints must be represented separately, again, with other UML diagrams.

Origin of Use Case

These days use case modeling is often associated with UML, although it has been introduced before UML existed. Its brief history is as follow:

- In 1986, Ivar Jacobson first formulated **textual** and **visual modeling** techniques for specifying use cases.
- In 1992 his co-authored book Object-Oriented Software Engineering - A Use Case Driven Approach helped to popularize the technique for capturing functional requirements, especially in software development.

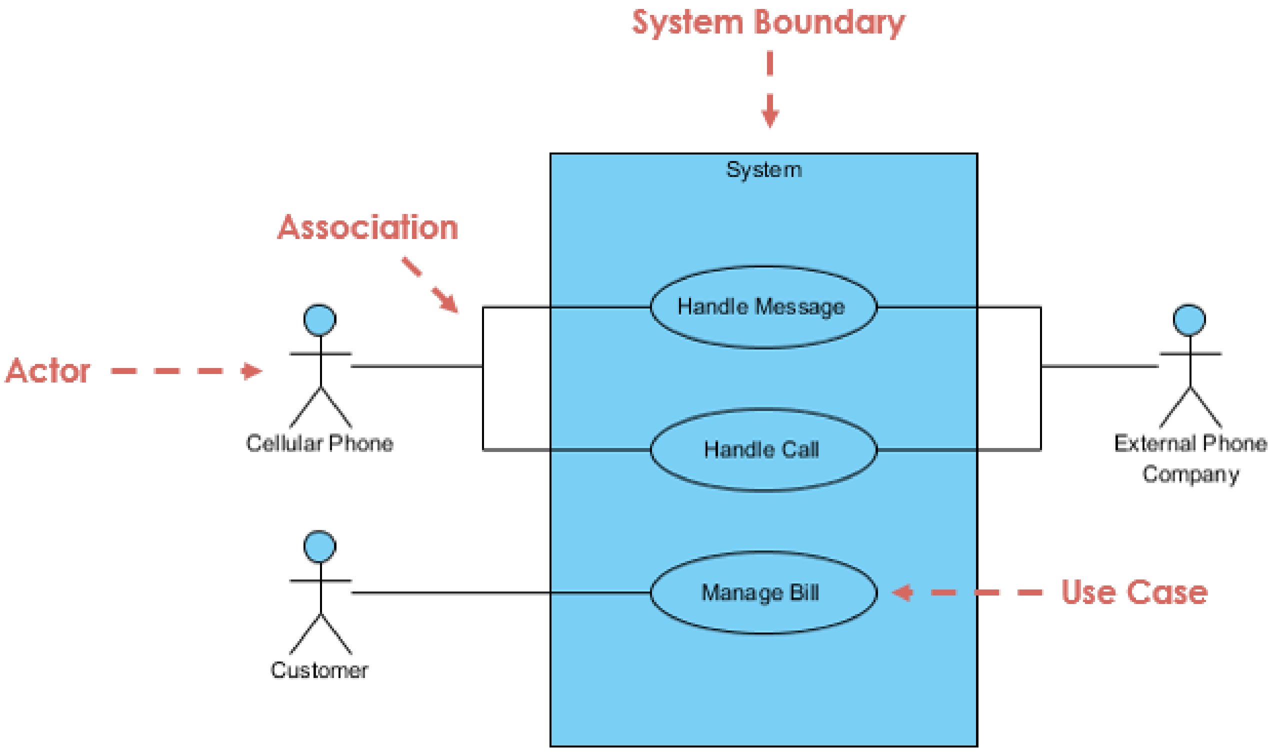
Purpose of Use Case Diagram

Use case diagrams are typically developed in the early stage of development and people often apply use case modeling for the following purposes:

- Specify the context of a system
- Capture the requirements of a system
- Validate a systems architecture
- Drive implementation and generate test cases
- Developed by analysts together with domain experts

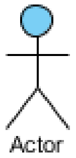
Use Case Diagram at a Glance

A standard form of use case diagram is defined in the Unified Modeling Language as shown in the Use Case Diagram example below:



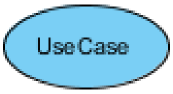
Notation Description	Visual Representation

Actor



- Someone interacts with use case (system function).
- Named by noun.
- Actor plays a role in the business
- Similar to the concept of user, but a user can play different roles
- For example:
  - A prof. can be instructor and also researcher
  - plays 2 roles with two systems
- Actor triggers use case(s).
- Actor has a responsibility toward the system (inputs), and Actor has expectations from the system (outputs).

Use Case

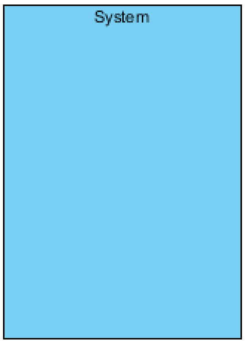


- System function (process - automated or manual)
- Named by verb + Noun (or Noun Phrase).
- i.e. Do something
- Each Actor must be linked to a use case, while some use cases may not be linked to actors.

Communication Link

- The participation of an actor in a use case is shown by connecting an actor to a use case by a solid link.
- Actors may be connected to use cases by associations, indicating that the actor and the use case communicate with one another using messages.

Boundary of system

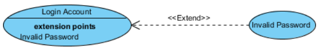
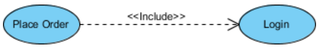
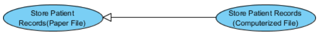


- The system boundary is potentially the entire system as defined in the requirements document.
- For large and complex systems, each module may be the system boundary.
- For example, for an ERP system for an organization, each of the modules such as personnel, payroll, accounting, etc.
- can form a system boundary for use cases specific to each of these business functions.
- The entire system can span all of these modules depicting the overall system boundary

Structuring Use Case Diagram with Relationships

Use cases share different kinds of relationships. Defining the relationship between two use cases is the decision of the software analysts of the use case diagram. A relationship between two use cases is basically modeling the dependency between the two use cases.

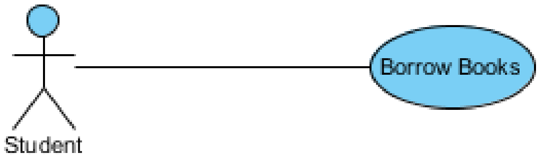
The reuse of an existing use case by using different types of relationships reduces the overall effort required in developing a system. Use case relationships are listed as the following:

Use Case Relationship	Visual Representation
Extends • Indicates that an **"Invalid Password"** use case may include (subject to specified in the extension) the behavior specified by base use case **"Login Account"**. • Depict with a directed arrow having a dotted line. The tip of arrowhead points to the base use case and the child use case is connected at the base of the arrow. • The stereotype "<<extends>>" identifies as an extend relationship	 Login Account extension points Invalid Password <<Extend>> Invalid Password
Include • When a use case is depicted as using the functionality of another use case, the relationship between the use cases is named as include or uses relationship. • A use case includes the functionality described in another use case as a part of its business process flow. • A uses relationship from base use case to child use case indicates that an instance of the base use case will include the behavior as specified in the child use case. • An include relationship is depicted with a directed arrow having a dotted line. The tip of arrowhead points to the child use case and the parent use case connected at the base of the arrow. • The stereotype "<<include>>" identifies the relationship as an include relationship.	
Generalization • A generalization relationship is a parent-child relationship between use cases. • The child use case is an enhancement of the parent use case. • Generalization is shown as a directed arrow with a triangle arrowhead. • The child use case is connected at the base of the arrow. The tip of the arrow is connected to the parent use case.	 Store Patient Records(Paper File) Store Patient Records (Computerized File)

Use Case Examples

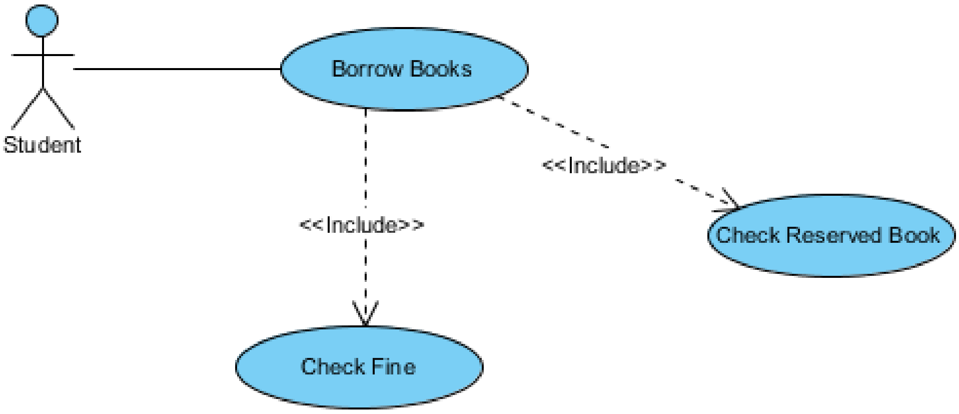
Use Case Example - Association Link

A Use Case diagram illustrates a set of use cases for a system, i.e. the actors and the relationships between the actors and use cases.



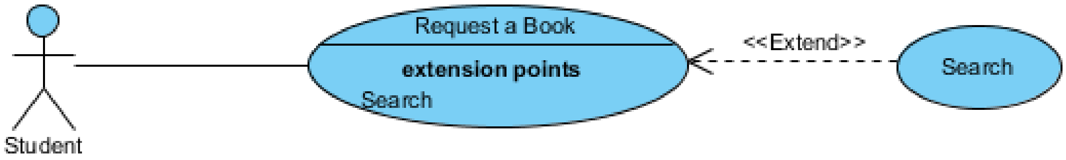
Use Case Example - Include Relationship

The include relationship adds additional functionality not specified in the base use case. The <<Include>> relationship is used to include common behavior from an included use case into a base use case in order to support the reuse of common behavior.



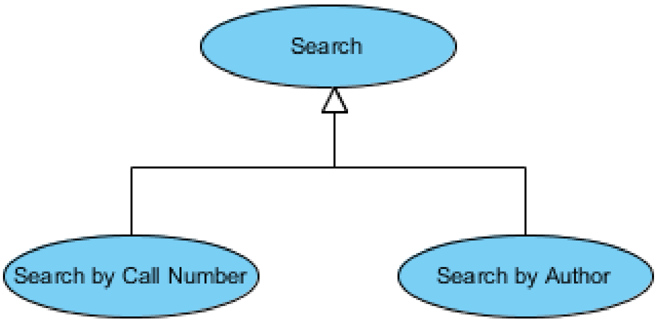
Use Case Example - Extend Relationship

The extend relationships are important because they show optional functionality or system behavior. The <<extend>> relationship is used to include optional behavior from an extending use case in an extended use case. Take a look at the use case diagram example below. It shows an extend connector and an extension point "Search".



Use Case Example - Generalization Relationship

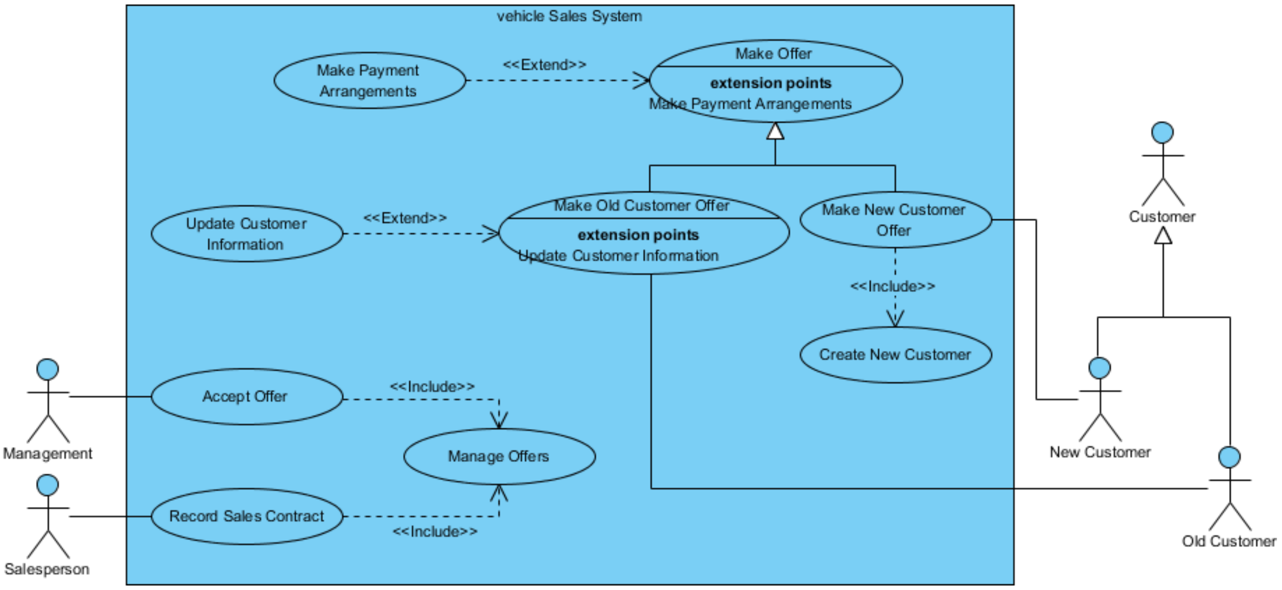
A generalization relationship means that a child use case inherits the behavior and meaning of the parent use case. The child may add or override the behavior of the parent. The figure below provides a use case example by showing two generalization connectors that connect between the three use cases.



Use Case Diagram - Vehicle Sales Systems

The figure below shows a use case diagram example for a vehicle system. As you can see even a system as big as a vehicle sales system contains not more than 10 use cases! That's the beauty of use case modeling.

The use case model also shows the use of extend and include. Besides, there are associations that connect between actors and use cases.



How to Identify Actor

Often, people find it easiest to start the requirements elicitation process by identifying the actors. The following questions can help you identify the actors of your system (Schneider and Winters - 1998):

- Who uses the system?
- Who installs the system?
- Who starts up the system?
- Who maintains the system?
- Who shuts down the system?
- What other systems use this system?
- Who gets information from this system?
- Who provides information to the system?

- Does anything happen automatically at a present time?

How to Identify Use Cases?

Identifying the Use Cases, and then the scenario-based elicitation process carries on by asking what externally visible, observable value that each actor desires. The following questions can be asked to identify use cases, once your actors have been identified (Schneider and Winters - 1998):

- What functions will the actor want from the system?
- Does the system store information? What actors will create, read, update or delete this information?
- Does the system need to notify an actor about changes in the internal state?
- Are there any external events the system must know about? What actor informs the system of those events?

Use Case Diagram Tips

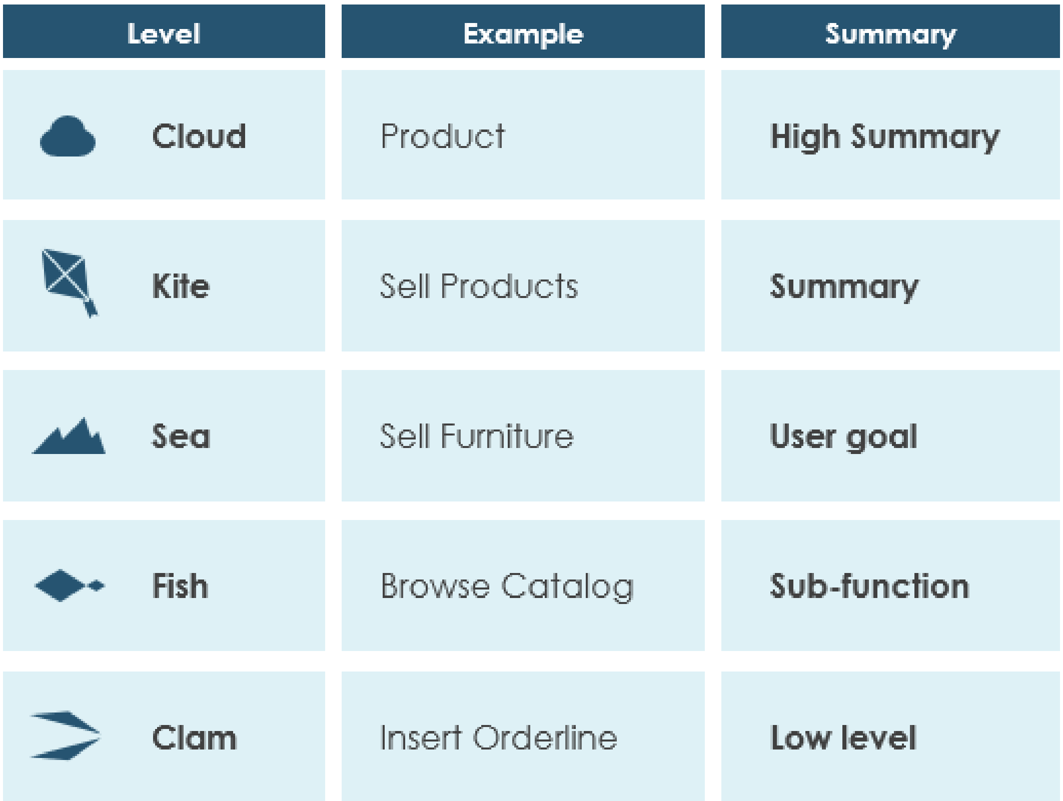
Now, check the tips below to see how to apply use case effectively in your software project.

- Always structure and organize the use case diagram from the perspective of actors.
- Use cases should start off simple and at the highest view possible. Only then can they be refined and detailed further.
- Use case diagrams are based upon functionality and thus should focus on the "what" and not the "how".

Use Case Levels of Details

Use case granularity refers to the way in which information is organized within use case specifications, and to some extent, the level of detail at which they are written. Achieving the right level of use case granularity eases communication between stakeholders and developers and improves project planning.

Alastair Cockburn in *Writing Effective Use Cases* gives us an easy way to visualize different levels of goal level by thinking in terms of the sea:



Level	Example	Summary
Cloud	Product	High Summary
Kite	Sell Products	Summary
Sea	Sell Furniture	User goal
Fish	Browse Catalog	Sub-function
Clam	Insert Orderline	Low level

Note that:

- While a use case itself might drill into a lot of detail about every possibility, a use-case diagram is often used for a higher-level view of the system as blueprints.
- It is beneficial to write use cases at a coarser level of granularity with less detail when it's not required.

What is Class Diagram?

visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/

In software engineering, a class diagram in the Unified Modeling Language (UML) is **a type of static structure diagram** that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

Purpose of Class Diagrams

1. Shows static structure of classifiers in a system
2. Diagram provides a basic notation for other structure diagrams prescribed by UML
3. Helpful for developers and other team members too
4. Business Analysts can use class diagrams to model systems from a business perspective

A UML class diagram is made up of:

- A set of classes and
- A set of relationships between classes

What is a Class

A description of a group of objects all with similar roles in the system, which consists of:

- **Structural features** (attributes) define what objects of the class "know"
  - Represent the state of an object of the class
  - Are descriptions of the structural or static features of a class
- **Behavioral features** (operations) define what objects of the class "can do"
  - Define the way in which objects may interact
  - Operations are descriptions of behavioral or dynamic features of a class

Class Notation

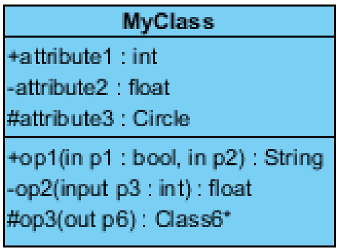
A class notation consists of three parts:

1. **Class Name**

   The name of the class appears in the first partition.
2. **Class Attributes**
   - Attributes are shown in the second partition.
   - The attribute type is shown after the colon.
   - Attributes map onto member variables (data members) in code.

3. **Class Operations** (Methods)
    - Operations are shown in the third partition. They are services the class provides.
    - The return type of a method is shown after the colon at the end of the method signature.
    - The return type of method parameters is shown after the colon following the parameter name.
    - Operations map onto class methods in code

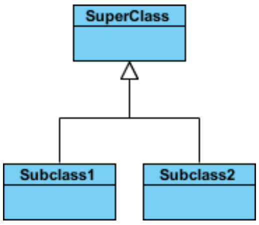
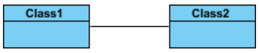
The graphical representation of the class - MyClass as shown above:



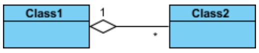
- MyClass has 3 attributes and 3 operations
- Parameter p3 of op2 is of type int
- op2 returns a float
- op3 returns a pointer (denoted by a *) to Class6

Class Relationships

A class may be involved in one or more relationships with other classes. A relationship can be one of the following types: (Refer to the figure on the right for the graphical representation of relationships).

Relationship Type	Graphical Representation
Inheritance (or Generalization): • Represents an "is-a" relationship. • An abstract class name is shown in italics. • SubClass1 and SubClass2 are specializations of Super Class. • A solid line with a hollow arrowhead that point from the child to the parent class	
Simple Association: • A structural link between two peer classes. • There is an association between Class1 and Class2 • A solid line connecting two classes	

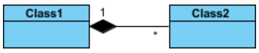
Aggregation:



A special type of association. It represents a "part of" relationship.

- Class2 is part of Class1.
- Many instances (denoted by the *) of Class2 can be associated with Class1.
- Objects of Class1 and Class2 have separate lifetimes.
- A solid line with an unfilled diamond at the association end connected to the class of composite

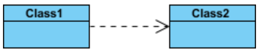
Composition:



A special type of aggregation where parts are destroyed when the whole is destroyed.

- Objects of Class2 live and die with Class1.
- Class2 cannot stand by itself.
- A solid line with a filled diamond at the association connected to the class of composite

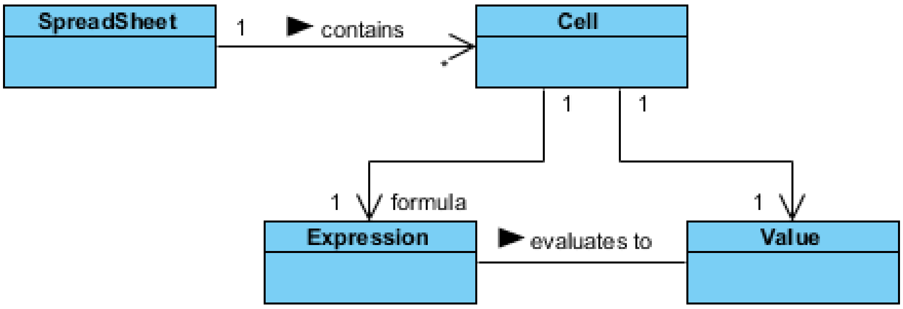
Dependency:



- Exists between two classes if the changes to the definition of one may cause changes to the other (but not the other way around).
- Class1 depends on Class2
- A dashed line with an open arrow

Relationship Names

- Names of relationships are written in the middle of the association line.
- Good relation names make sense when you read them out loud:
  - "Every spreadsheet **contains** some number of cells",
  - "an expression **evaluates to** a value"
- They often have a **small arrowhead to show the direction** in which direction to read the relationship, e.g., expressions evaluate to values, but values do not evaluate to expressions.



Relationship - Roles

- A role is a directional purpose of an association.
- Roles are written at the ends of an association line and describe the purpose played by that class in the relationship.
  E.g., A cell is related to an expression. The nature of the relationship is that the expression is the **formula** of the cell.

Navigability

The arrows indicate whether, given one instance participating in a relationship, it is possible to determine the instances of the other class that are related to it.

The diagram above suggests that,

- Given a spreadsheet, we can locate all of the cells that it contains, but that
  we cannot determine from a cell in what spreadsheet it is contained.
- Given a cell, we can obtain the related expression and value, but
  given a value (or expression) we cannot find the cell of which those are attributes.

Visibility of Class attributes and Operations

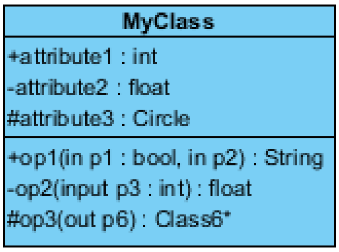
In object-oriented design, there is a notation of visibility for attributes and operations. UML identifies four types of visibility: **public**, **protected**, **private**, and **package**.

The +, -, # and ~ symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.

- + denotes public attributes or operations
- - denotes private attributes or operations
- # denotes protected attributes or operations
- ~ denotes package attributes or operations

Class Visibility Example

In the example above:



- attribute1 and op1 of MyClassName are public
- attribute3 and op3 are protected.
- attribute2 and op2 are private.

Access for each of these visibility types is shown below for members of different classes.

Access Right	public (+)	private (-)	protected (#)	Package (~)
Members of the same class	yes	yes	yes	yes

Members of derived classes	yes	no	yes	yes
Members of any other class	yes	no	no	in same package

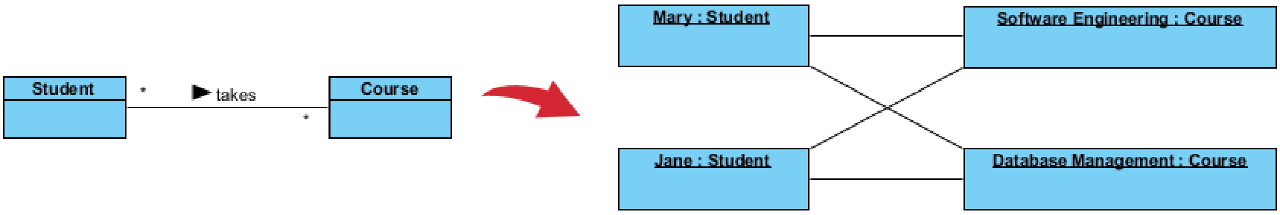
Multiplicity

How many objects of each class take part in the relationships and multiplicity can be expressed as:

- Exactly one - 1
- Zero or one - 0..1
- Many - 0..* or *
- One or more - 1..*
- Exact Number - e.g. 3..4 or 6
- Or a complex relationship - e.g. 0..1, 3..4, 6.* would mean any number of objects other than 2 or 5

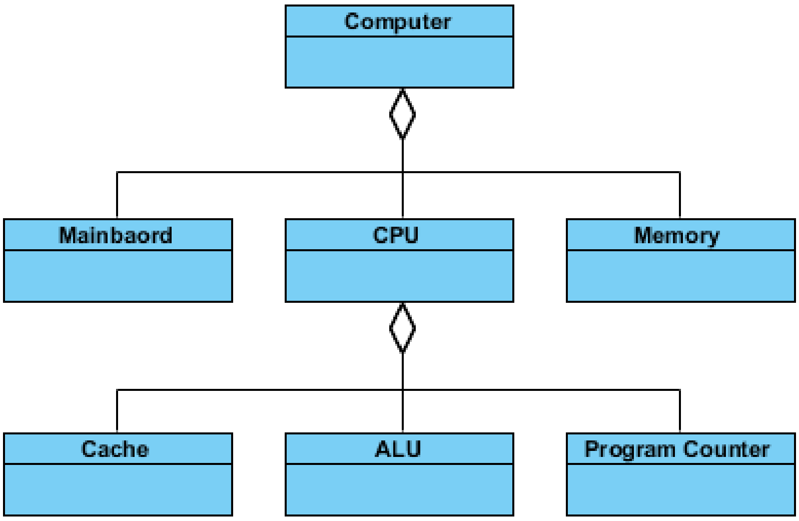
Multiplicity Example

- Requirement: A Student can take many Courses and many Students can be enrolled in one Course.
- In the example below, the **class diagram** (on the left), describes the statement of the requirement above for the static model while the object diagram (on the right) shows the snapshot (an instance of the class diagram) of the course enrollment for the courses Software Engineering and Database Management respectively)



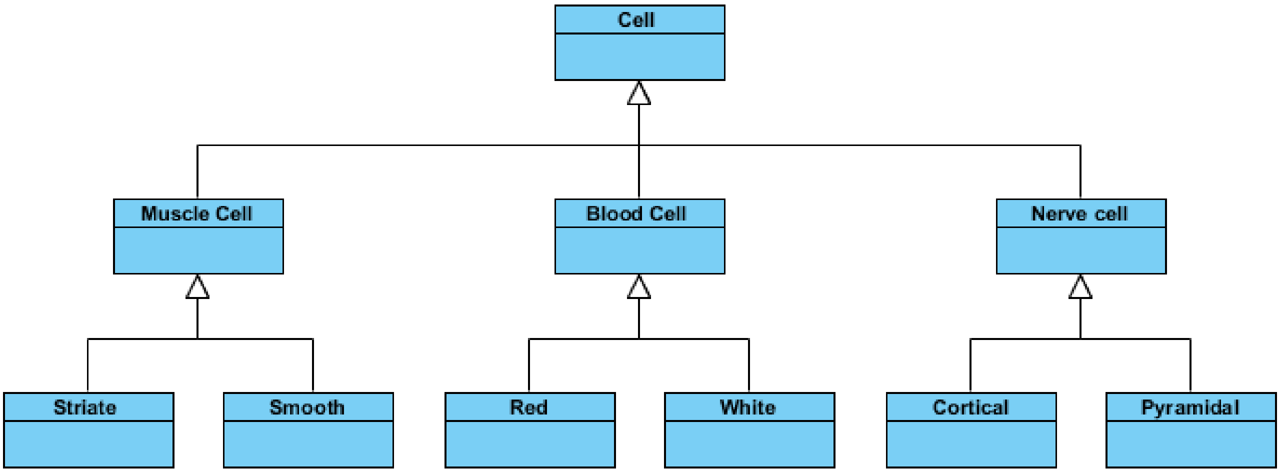
Aggregation Example - Computer and parts

- An aggregation is a special case of association denoting a "consists-of" hierarchy
- The aggregate is the parent class, the components are the children classes



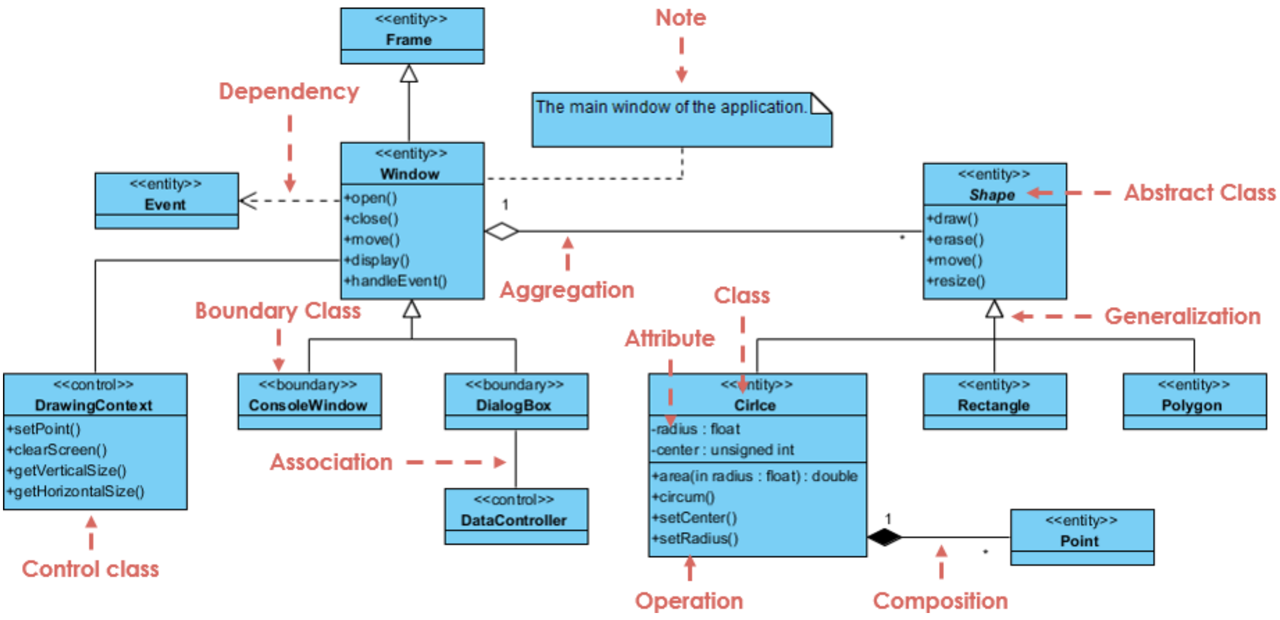
Inheritance Example - Cell Taxonomy

- Inheritance is another special case of an association denoting a "kind-of" hierarchy
- Inheritance simplifies the analysis model by introducing a taxonomy
- The child classes inherit the attributes and operations of the parent class.



Class Diagram - Diagram Tool Example

A class diagram may also have notes attached to classes or relationships. Notes are shown in grey.



In the example above:

We can interpret the meaning of the above class diagram by reading through the points as following.

1. Shape is an abstract class. It is shown in Italics.
2. Shape is a superclass. Circle, Rectangle and Polygon are derived from Shape. In other words, a Circle is-a Shape. This is a generalization / inheritance relationship.
3. There is an association between DialogBox and DataController.
4. Shape is part-of Window. This is an aggregation relationship. Shape can exist without Window.
5. Point is part-of Circle. This is a composition relationship. Point cannot exist without a Circle.
6. Window is dependent on Event. However, Event is not dependent on Window.
7. The attributes of Circle are radius and center. This is an entity class.
8. The method names of Circle are area(), circum(), setCenter() and setRadius().
9. The parameter radius in Circle is an in parameter of type float.
10. The method area() of class Circle returns a value of type double.
11. The attributes and method names of Rectangle are hidden. Some other classes in the diagram also have their attributes and method names hidden.

Dealing with Complex System - Multiple or Single Class Diagram?

Inevitably, if you are modeling a large system or a large business area, there will be numerous entities you must consider. Should we use multiple or a single class diagram for modeling the problem? The answer is:

- Instead of modeling every entity and its relationships on a single class diagram, it is better to use multiple class diagrams.

- Dividing a system into multiple class diagrams makes the system easier to understand, especially if each diagram is a graphical representation of a specific part of the system.

Perspectives of Class Diagram in Software Development Lifecycle

We can use class diagrams in different development phases of a software development lifecycle and typically by modeling class diagrams in three different perspectives (levels of detail) progressively as we move forward:

Conceptual perspective: The diagrams are interpreted as describing things in the real world. Thus, if you take the conceptual perspective you draw a diagram that represents the concepts in the domain under study. These concepts will naturally relate to the classes that implement them. The conceptual perspective is **considered language-independent**.

Specification perspective: The diagrams are interpreted as describing software abstractions or components with specifications and interfaces but with no commitment to a particular implementation. Thus, if you take the specification perspective we are **looking at the interfaces of the software**, not the implementation.

Implementation perspective: The diagrams are interpreted as describing software implementations in a particular technology and language. Thus, if you take the implementation perspective we are **looking at the software implementation**.

What is State Machine Diagram?

visual-paradigm.com/guide/uml-unified-modeling-language/what-is-state-machine-diagram/

The behavior of an entity is not only a direct consequence of its inputs, but it also depends on its preceding state. The past history of an entity can best be modeled by a finite state machine diagram or traditionally called automata. UML State Machine Diagrams (or sometimes referred to as state diagram, state machine or state chart) show the different states of an entity. State machine diagrams can also show how an entity responds to various events by changing from one state to another. State machine diagram is a UML diagram used to model the dynamic nature of a system.

Why State Machine Diagrams?

State machine diagram typically are used to describe state-dependent behavior for an object. **An object responds differently to the same event depending on what state it is in**. State machine diagrams are usually applied to objects but can be applied to any element that has behavior to other entities such as: actors, use cases, methods, subsystems systems and etc. and they are typically used in conjunction with interaction diagrams (usually sequence diagrams).

For example:

Consider you have $100,000 in a bank account. The behavior of the withdraw function would be: balance := balance - withdrawAmount; provided that **the balance after the withdrawal is not less than $0;** this is true regardless of how many times you have withdrawn money from the bank. In such situations, the withdrawals do not affect the abstraction of the attribute values, and hence the gross behavior of the object remains unchanged.

However, if the **account balance would become negative after a withdrawal**, the behavior of the withdraw function would be quite different. This is because the state of the bank account is changed from positive to negative; in technical jargon, a transition from the positive state to the negative state is fired.

The abstraction of the attribute value is a property of the system, rather than a globally applicable rule. For example, if the bank changes the business rule to allow the bank balance to be overdrawn by 2000 dollars, the state of the bank account will be redefined with condition that the balance after withdrawal must not be less than $2000 in deficit.

Note That:

- A state machine diagram describes all events (and states and transitions for a single object)
- A sequence diagram describes the events for a single interaction across all objects involved

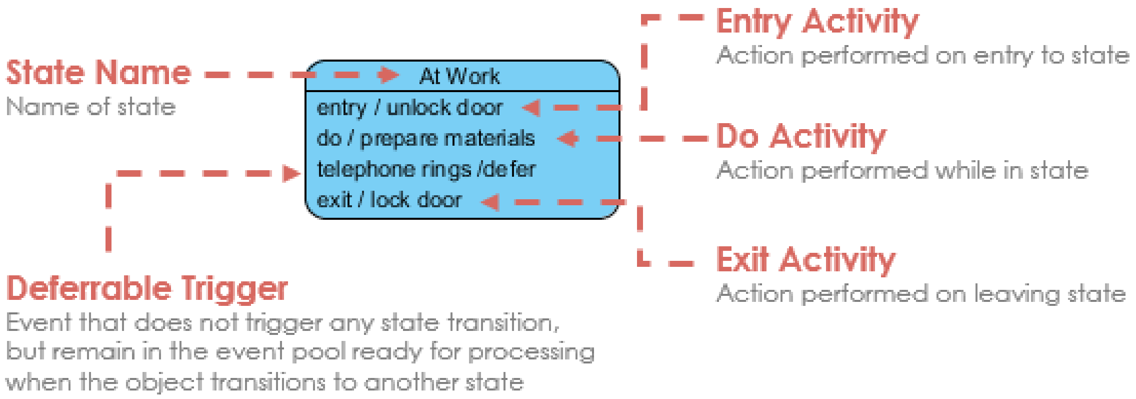
Basic Concepts of State Machine Diagram

What is a State?

Rumbaugh defines that:

"A state is an abstraction of the attribute values and links of an object. Sets of values are grouped together into a state according to properties that affect the gross behavior of the object."

State Notation



State Name
Name of state
At Work
entry / unlock door
do / prepare materials
telephone rings /defer
exit / lock door
Entry Activity
Action performed on entry to state
Do Activity
Action performed while in state
Exit Activity
Action performed on leaving state
Deferrable Trigger
Event that does not trigger any state transition, but remain in the event pool ready for processing when the object transitions to another state

Characteristics of State Machine Notations

There are several characteristics of states in general, regardless of their types:

- A state occupies an interval of time.
- A state is often associated with an abstraction of attribute values of an entity satisfying some condition(s).
- An entity changes its state not only as a direct consequence of the current input, but it is also dependent on some past history of its inputs.

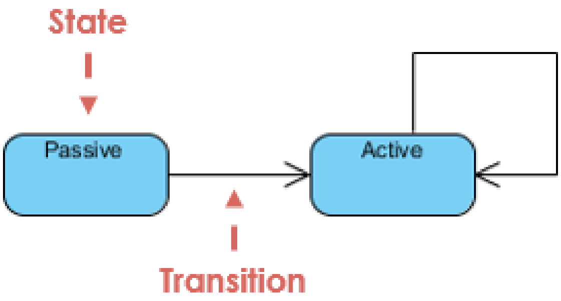
State

A state is a constraint or a situation in the life cycle of an object, in which a constraint holds, the object executes an activity or waits for an event.

A state machine diagram is a graph consisting of:

- States (simple states or composite states)
- State transitions connecting the states

Example:



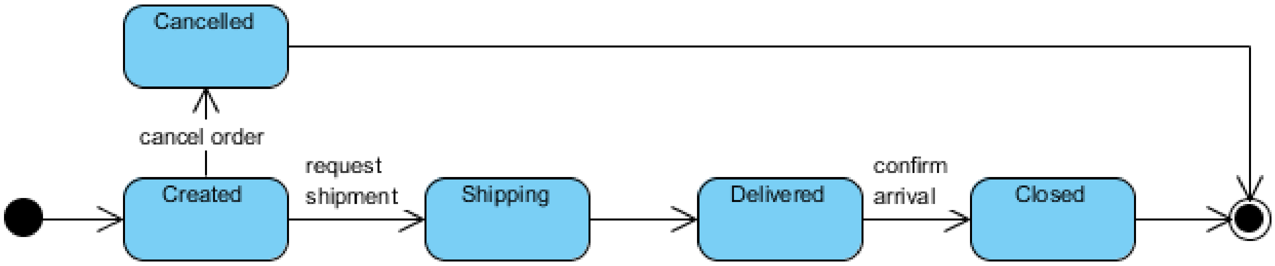
Characteristics of State

- State represent the conditions of objects at certain points in time.
- Objects (or Systems) can be viewed as moving from state to state
- A point in the lifecycle of a model element that satisfies some condition, where some particular action is being performed or where some event is waited

Initial and Final States

- The **initial state** of a state machine diagram, known as an initial pseudo-state, is indicated with a solid circle. A transition from this state will show the first real state
- The **final state** of a state machine diagram is shown as concentric circles. An open loop state machine represents an object that may terminate before the system terminates, while a closed loop state machine diagram does not have a final state; if it is the case, then the object lives until the entire system terminates.

Example:



Events

An event signature is described as Event-name (comma-separated-parameter-list). Events appear in the internal transition compartment of a state or on a transition between states. An event may be one of four types:

1. Signal event - corresponding to the arrival of an asynchronous message or signal
2. Call event - corresponding to the arrival of a procedural call to an operation
3. Time event - a time event occurs after a specified time has elapsed
4. Change event - a change event occurs whenever a specified condition is met

Characteristics of Events

- Represents incidents that cause objects to transition from one state to another.

- Internal or External Events trigger some activity that changes the state of the system and of some of its parts
- Events pass information, which is elaborated by Objects operations. Objects realize Events
- Design involves examining events in a state machine diagram and considering how those events will be supported by system objects

Transition

Transition lines depict the movement from one state to another. Each transition line is labeled with the **event** that causes the transition.

- Viewing a system as a set of states and transitions between states is very useful for describing complex behaviors
- Understanding state transitions is part of system analysis and design
- A Transition is the movement from one state to another state
- Transitions between states occur as follows:
    1. An element is in a source state
    2. An event occurs
    3. An action is performed
    4. The element enters a target state
- Multiple transitions occur either when different events result in a state terminating or when there are guard conditions on the transitions
- A transition without an event and action is known as automatic transitions

Actions

Action is an executable atomic computation, which includes operation calls, the creation or destruction of another object, or the sending of a signal to an object. An action is associated with transitions and during which an action is not interruptible - e.g., entry, exit

Activity

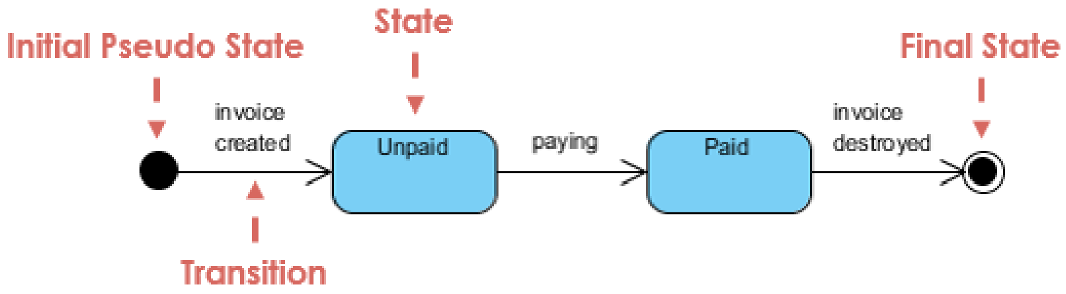
Activity is associated with states, which is a non-atomic or ongoing computation. Activity may run to completion or continue indefinitely. An Activity will be terminated by an event that causes a transition from the state in which the activity is defined

Characteristics of Action and Activities

- States can trigger actions
- States can have a second compartment that contains actions or activities performed while an entity is in a given state
- An action is an atomic execution and therefore completes without interruption
- Five triggers for actions: On Entry, Do, On Event, On Exit, and Include

- An activity captures complex behavior that may run for a long duration - An activity may be interrupted by events, in which case it does not complete occur when an object arrives in a state.

Simple State Machine Diagram Notation



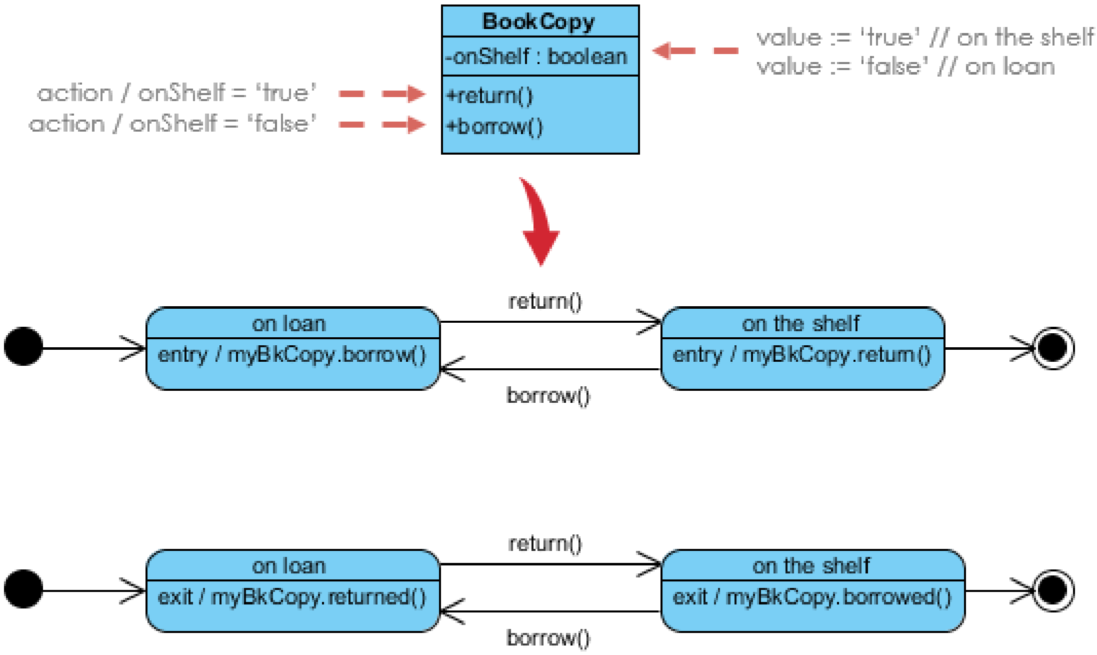
Entry and Exit Actions

Entry and Exit actions specified in the state. It must be true for every entry / exit occurrence. If not, then you must use actions on the individual transition arcs

- **Entry Action** executed on entry into state with the **notation: Entry / action**
- **Exit Action** executed on exit from state with the **notation: Exit / action**

Example - Entry / Exit Action (Check Book Status)

This example illustrates a state machine diagram derived from a Class - "BookCopy":


BookCopy
-onShelf : boolean
+return()
+borrow()
value := 'true' // on the shelf
value := 'false' // on loan
action / onShelf = 'true'
action / onShelf = 'false'
on loan
entry / myBkCopy.borrow()
return()
borrow()
on the shelf
entry / myBkCopy.return()
on loan
exit / myBkCopy.returned()
return()
borrow()
on the shelf
exit / myBkCopy.borrowed()

Note:

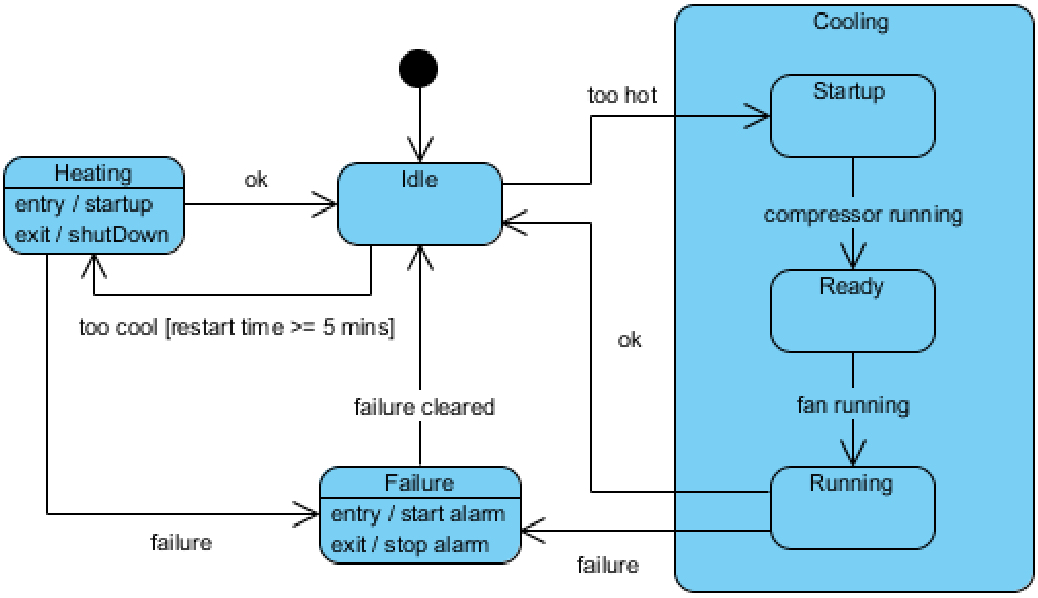
1. This state machine diagram shows the state of an object myBkCopy from a BookCopy class
2. Entry action : any action that is marked as linked to the entry action is executed whenever the given state is entered via a transition

3. Exit action : any action that is marked as linked to the exit action is executed whenever the state is left via a transition

Substates

A simple state is one which has no substructure. A state which has substates (nested states) is called a composite state. Substates may be nested to any level. A nested state machine may have at most one initial state and one final state. Substates are used to simplify complex flat state machines by showing that some states are only possible within a particular context (the enclosing state).

Substate Example - Heater



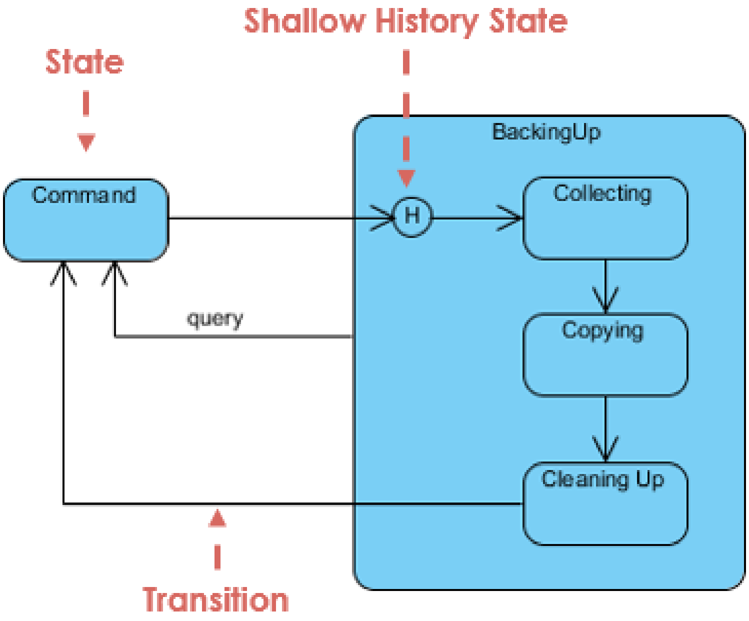
State Machine Diagrams are often used for deriving testing cases, here is a list of possible test ideas:

- Idle state receives Too Hot event
- Idle state receives Too Cool event
- Cooling/Startup state receives Compressor Running event
- Cooling/Ready state receives Fan Running event
- Cooling/Running state receives OK event
- Cooling/Running state receives Failure event
- Failure state receives Failure Cleared event
- Heating state receives OK event
- Heating state receives Failure event

History States

Unless otherwise specified, when a transition enters a composite state, the action of **the nested state machine starts over again at the initial state** (unless the transition targets a substate directly). History states allow the state machine to **re-enter the last**

substate that was active prior to leaving the composite state. An example of history state usage is presented in the figure below.

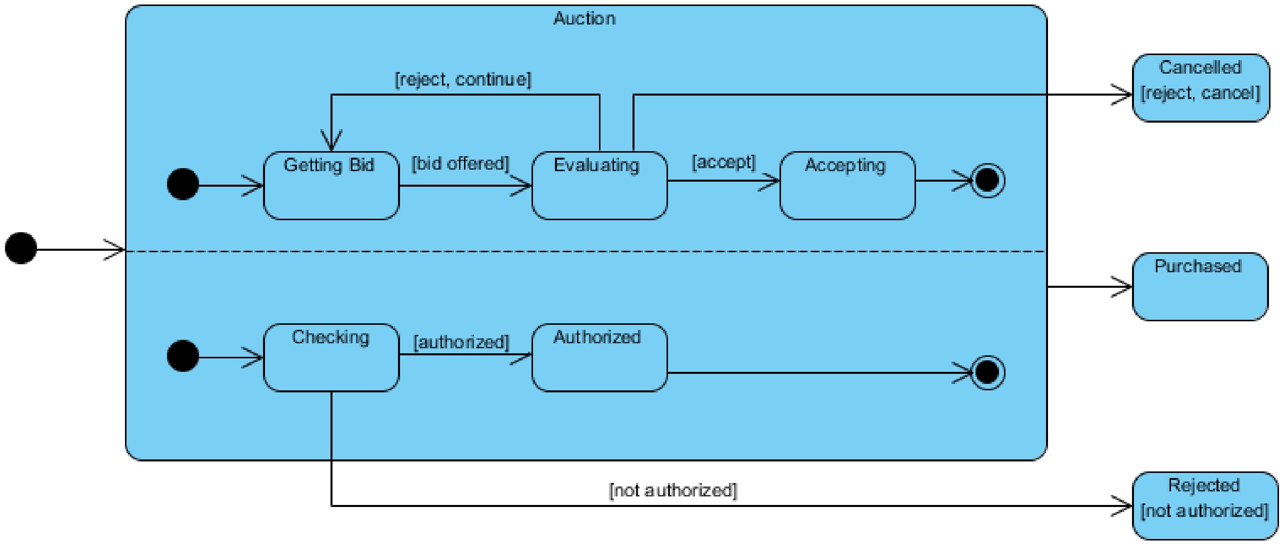


Concurrent State

As mentioned above, states in state machine diagrams can be nested. Related states can be grouped together into a single composite state. Nesting states inside others is necessary when an activity involves concurrent sub-activities. The following state machine diagram models an auction with two concurrent substates: processing the bid and authorizing the payment limit.

Concurrent State Machine Diagram Example - Auction Process

In this example, the state machine first entering the Auction requires a fork at the start into two separate start threads. Each substate has an exit state to mark the end of the thread. Unless there is an abnormal exit (Canceled or Rejected), the exit from the composite state occurs when both substates have exited.



What is Sequence Diagram?

visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram/

UML Sequence Diagrams are interaction diagrams that detail how operations are carried out. They capture the interaction between objects in the context of a collaboration. Sequence Diagrams are time focus and they show the order of the interaction visually by using the vertical axis of the diagram to represent time what messages are sent and when.

Sequence Diagrams captures:

- the interaction that takes place in a collaboration that either realizes a use case or an operation (instance diagrams or generic diagrams)
- high-level interactions between user of the system and the system, between the system and other systems, or between subsystems (sometimes known as system sequence diagrams)

Purpose of Sequence Diagram

- Model high-level interaction between active objects in a system
- Model the interaction between object instances within a collaboration that realizes a use case
- Model the interaction between objects within a collaboration that realizes an operation
- Either model generic interactions (showing all possible paths through the interaction) or specific instances of a interaction (showing just one path through the interaction)

Sequence Diagrams at a Glance

Sequence Diagrams show elements as they interact over time and they are organized according to object (horizontally) and time (vertically):

Object Dimension

- The horizontal axis shows the elements that are involved in the interaction
- Conventionally, the objects involved in the operation are listed from left to right according to when they take part in the message sequence. However, the elements on the horizontal axis may appear in any order

Time Dimension

The vertical axis represents time proceedings (or progressing) down the page.

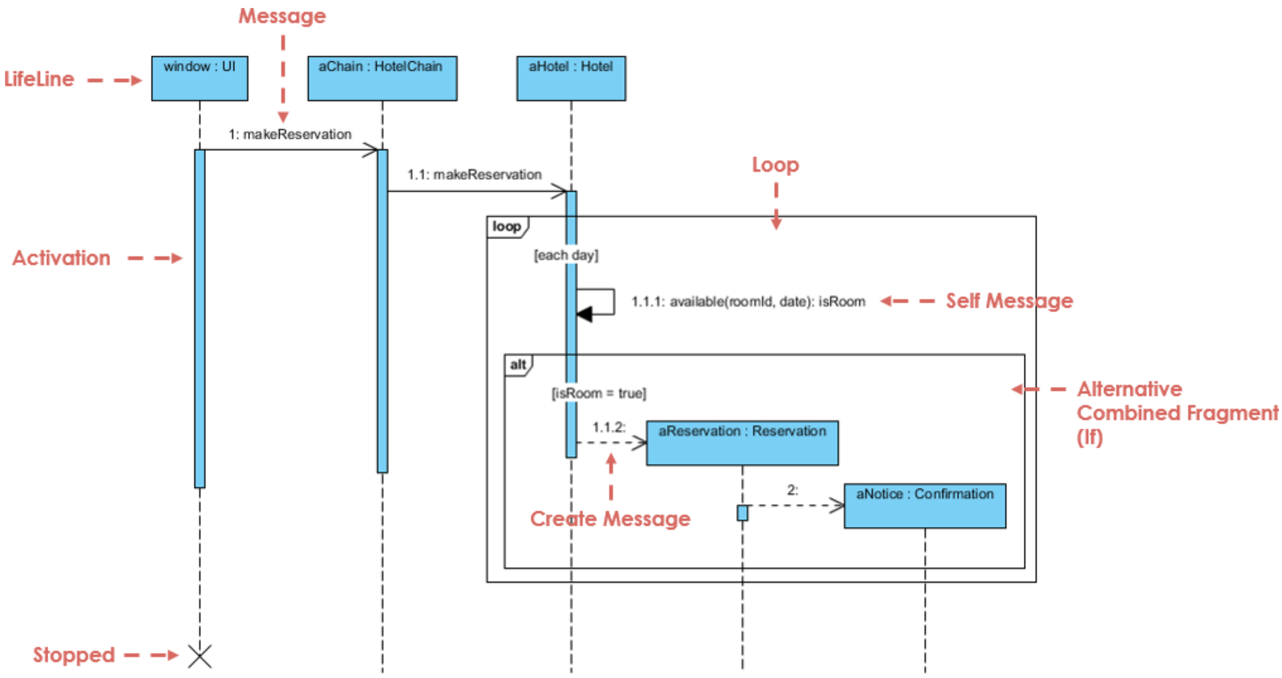
Note that:

Time in a sequence diagram is all a about ordering, not duration. The vertical space in an interaction diagram is not relevant for the duration of the interaction.

Sequence Diagram Example: Hotel System

Sequence Diagram is an interaction diagram that details how operations are carried out -- what messages are sent and when. Sequence diagrams are organized according to time. The time progresses as you go down the page. The objects involved in the operation are listed from left to right according to when they take part in the message sequence.

Below is a sequence diagram for making a hotel reservation. The object initiating the sequence of messages is a Reservation window.

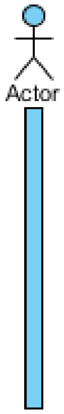


Note That: Class and object diagrams are static model views. Interaction diagrams are dynamic. They describe how objects collaborate.

Sequence Diagram Notation

Notation Description	Visual Representation

Actor

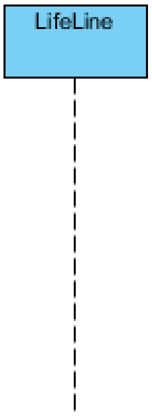


- a type of role played by an entity that interacts with the subject (e.g., by exchanging signals and data)
- external to the subject (i.e., in the sense that an instance of an actor is not a part of the instance of its corresponding subject).
- represent roles played by human users, external hardware, or other subjects.

Note that:

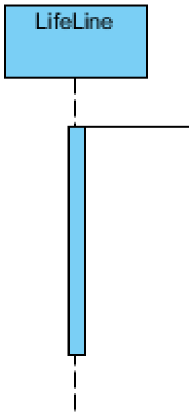
- An actor does not necessarily represent a specific physical entity but merely a particular role of some entity
- A person may play the role of several different actors and, conversely, a given actor may be played by multiple different person.

Lifeline



A lifeline represents an individual participant in the Interaction.

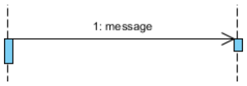
Activations



- A thin rectangle on a lifeline) represents the period during which an element is performing an operation.
- The top and the bottom of the of the rectangle are aligned with the initiation and the completion time respectively

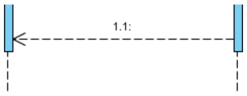
Call Message

- A message defines a particular communication between Lifelines of an Interaction.
- Call message is a kind of message that represents an invocation of operation of target lifeline.



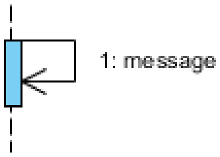
Return Message

- A message defines a particular communication between Lifelines of an Interaction.
- Return message is a kind of message that represents the pass of information back to the caller of a corresponded former message.



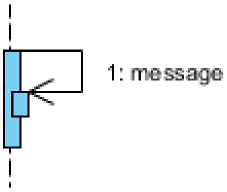
Self Message

- A message defines a particular communication between Lifelines of an Interaction.
- Self message is a kind of message that represents the invocation of message of the same lifeline.



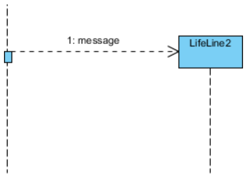
Recursive Message

- A message defines a particular communication between Lifelines of an Interaction.
- Recursive message is a kind of message that represents the invocation of message of the same lifeline. It's target points to an activation on top of the activation where the message was invoked from.



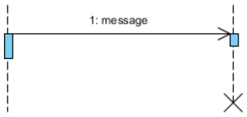
Create Message

- A message defines a particular communication between Lifelines of an Interaction.
- Create message is a kind of message that represents the instantiation of (target) lifeline.

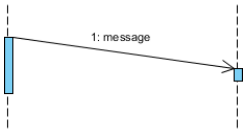


Destroy Message

- A message defines a particular communication between Lifelines of an Interaction.
- Destroy message is a kind of message that represents the request of destroying the lifecycle of target lifeline.

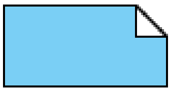


Duration Message



- A message defines a particular communication between Lifelines of an Interaction.
- Duration message shows the distance between two time instants for a message invocation.

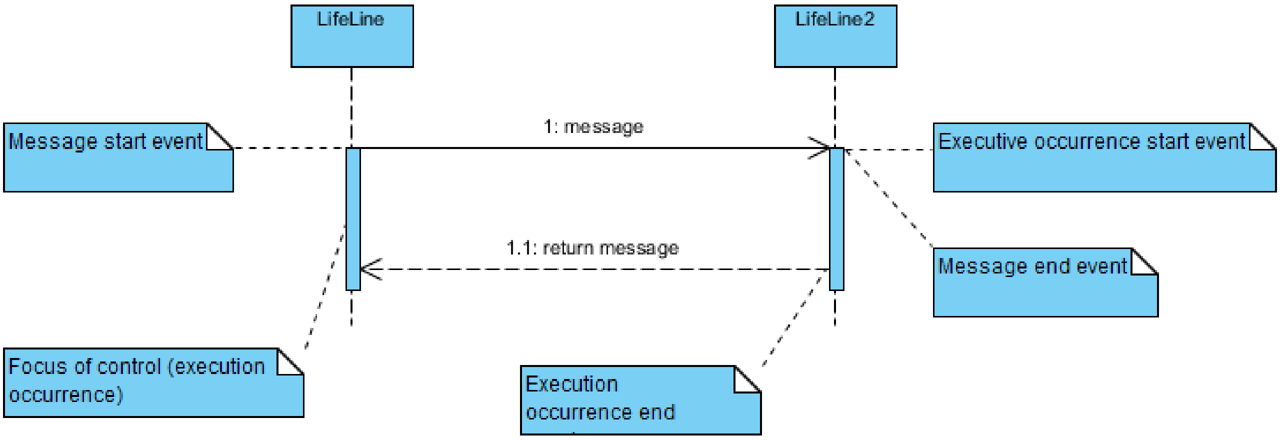
Note



A note (comment) gives the ability to attach various remarks to elements. A comment carries no semantic force, but may contain information that is useful to a modeler.

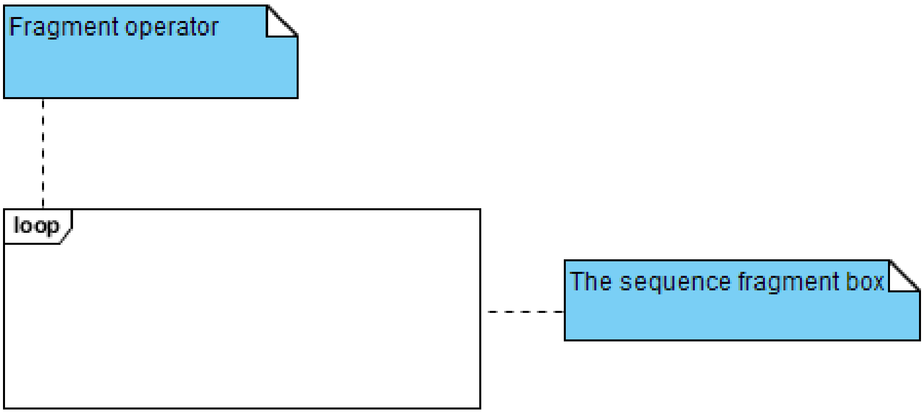
Message and Focus of Control

- An Event is any point in an interaction where something occurs.
- Focus of control: also called execution occurrence, an execution occurrence
- It shows as tall, thin rectangle on a lifeline)
- It represents the period during which an element is performing an operation. The top and the bottom of the rectangle are aligned with the initiation and the completion time respectively.



Sequence Fragments

- UML 2.0 introduces sequence (or interaction) fragments. Sequence fragments make it easier to create and maintain accurate sequence diagrams
- A sequence fragment is represented as a box, called a combined fragment, which encloses a portion of the interactions within a sequence diagram
- The fragment operator (in the top left cornet) indicates the type of fragment
- Fragment types: ref, assert, loop, break, alt, opt, neg

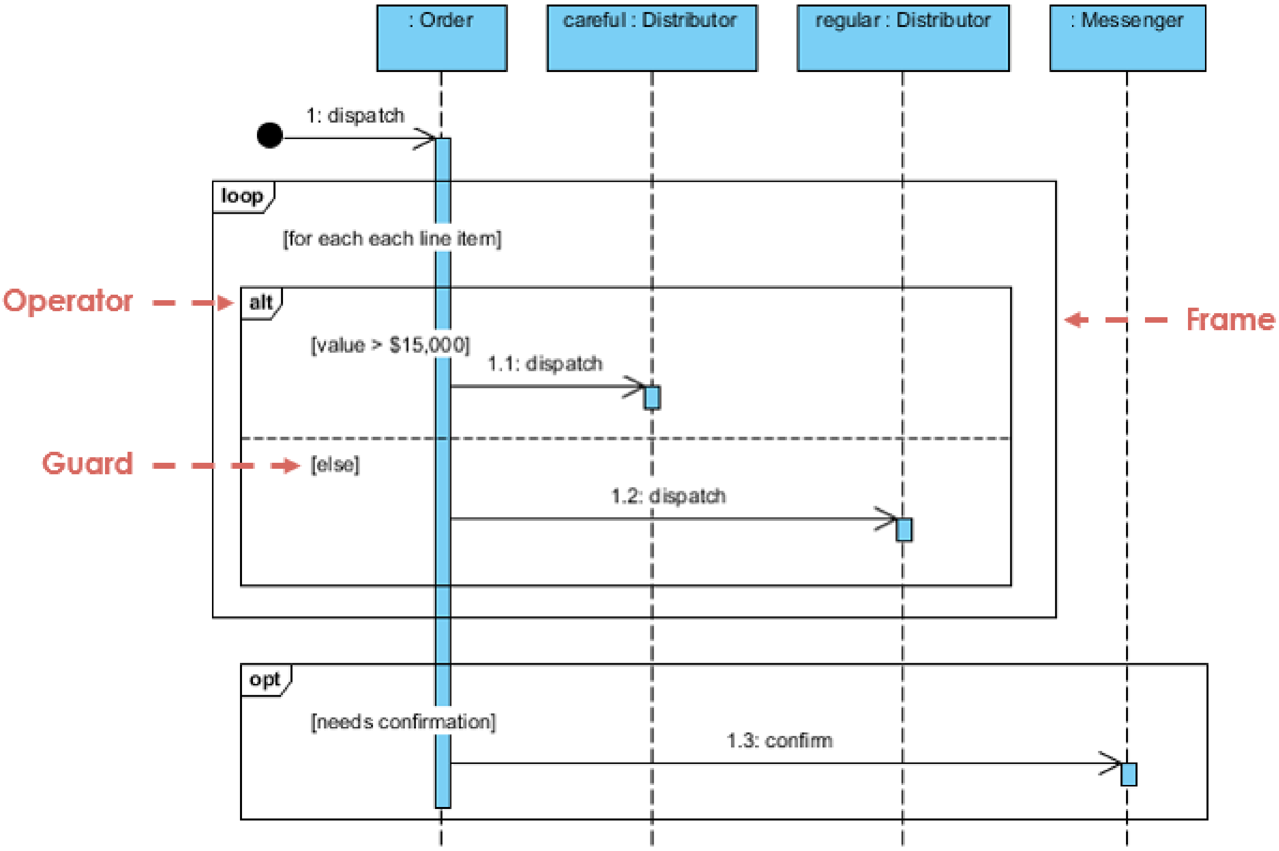


Operator	Fragment Type
alt	Alternative multiple fragments: only the one whose condition is true will execute.
opt	Optional: the fragment executes only if the supplied condition is true. Equivalent to an alt only with one trace.
par	Parallel: each fragment is run in parallel.
loop	Loop: the fragment may execute multiple times, and the guard indicates the basis of iteration.
region	Critical region: the fragment can have only one thread executing it at once.
neg	Negative: the fragment shows an invalid interaction.
ref	Reference: refers to an interaction defined on another diagram. The frame is drawn to cover the lifelines involved in the interaction. You can define parameters and a return value.
sd	Sequence diagram: used to surround an entire sequence diagram.

Note That:

- It is possible to combine frames in order to capture, e.g., loops or branches.
- **Combined fragment** keywords: alt, opt, break, par, seq, strict, neg, critical, ignore, consider, assert and loop.
- Constraints are usually used to show timing constraints on messages. They can apply to the timing of one message or intervals between messages.

Combined Fragment Example

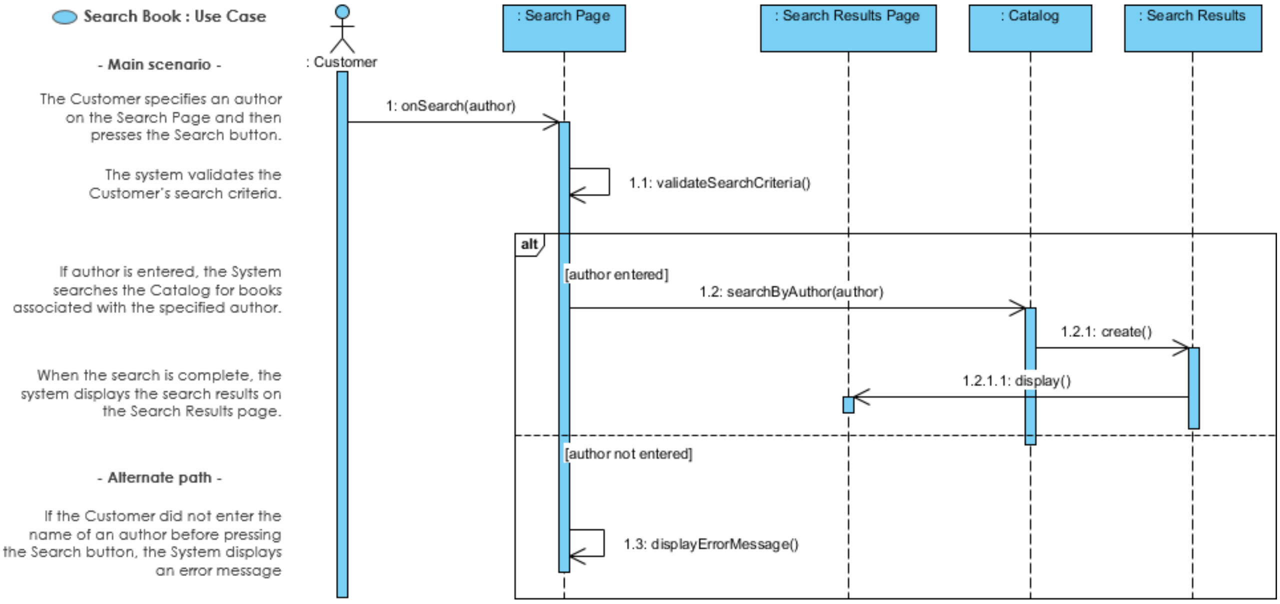


Sequence Diagram for Modeling Use Case Scenarios

User requirements are captured as use cases that are refined into scenarios. A use case is a collection of interactions between external actors and a system. In UML, a use case is:

"the specification of a sequence of actions, including variants, that a system (or entity) can perform, interacting with actors of the system."

A scenario is one path or flow through a use case that describes a sequence of events that occurs during one particular execution of a system which is often represented by a sequence diagram.


Search Book : Use Case
- Main scenario -
The Customer specifies an author on the Search Page and then presses the Search button.
The system validates the Customer's search criteria.
If author is entered, the System searches the Catalog for books associated with the specified author.
When the search is complete, the system displays the search results on the Search Results page.
- Alternate path -
If the Customer did not enter the name of an author before pressing the Search button, the System displays an error message
: Customer
: Search Page
: Search Results Page
: Catalog
: Search Results
1: onSearch(author)
1.1: validateSearchCriteria()
alt
[author entered]
1.2: searchByAuthor(author)
1.2.1: create()
1.2.1.1: display()
[author not entered]
1.3: displayErrorMessage()

Sequence Diagram - Model before Code

Sequence diagrams can be somewhat close to the code level, so why not just code up that algorithm rather than drawing it as a sequence diagram?

- A good sequence diagram is still a bit above the level of the real code
- Sequence diagrams are language neutral
- Non-coders can do sequence diagrams
- Easier to do sequence diagrams as a team
- Can be used for testing and/or UX Wireframing